Comparison of the performance of parallelizing the K-Means algorithm using MPI, OMP, and a hybrid version

High Performance Computing (HPC) Project

Edoardo Maines (232226) & Michele Presutto (229208)

February 19, 2024

University of Trento

1 Introduction

In a world where data production is increasing day by day the need for data analytic method to extract information from the vast disposition of data it's growing faster. Among the technique used for analyse data, clustering is one of the most used. The objective of clustering is to subdivide a dataset into subcategories based on the data characteristics, however analizing a large quantity of data can be challenging in terms of execution time and resources. To cope for this problem there are techniques to parallelize the execution of these algorithms and improve performances.

in this paper we present the implementation of a clustering algorithm written in C language, using MPI (Message Passing Interface) and OMP(Open Multi-Processing) to parallelize the code.
Our scope is to compare the performances between the serial and the parallel executions. Particularly we will compare the execution time and the precision of the different implementations of the algorithms(`OMP`,`MPI`) to better understand which one is more efficient. In the end we finalised a third implementation that consist of an hybrid approach(MPI+OMP) to observe if the union of both methodology could improve the performances.
Our work it's been implemented and tested using the server cluster belonging to the university of Trento.

2 State of the Art

clustering is a data analysis technique widely used in a lot of disciplines as biology, computer science, statistics and much more. There are different kind of clustering algorithms among which hierarchical clustering, k-means clustering and Gaussian Mixture Model clustering.

hierarchic clustering subdivide the data in hierarchic groups applying subsequential division till each group has only one element building this way a dendogram to represent the data distribuition and possible clusters as shown in [3].

K-means clustering is a partitioning technique that attempts to divide the data into a fixed number of groups, so that the variance within the groups is minimal, while Mixed model clustering is a technique that uses probability models to associate data with groups.

Some clustering algorithms were designed for clustering large amounts of data. For example, clustering based on grid patterns divides the data space into a grid, where each cell represents a group. This technique is particularly useful for analyzing geospatial data.

we analysed different state of the art algorithms for parallelizing the k-means with MPI[4] [**Yang**] and OMP [2] [1].

3 Parallelization methods

3.1 MPI (Message Passing Interface)

MPI is a parallel programming library widely used to develop distributed and parallel applications on computer clusters. It enables communication between processes, allowing them to exchange data and coordinate operations synchronously or asynchronously. The main features of MPI include message management, synchronization, data distribution, and scalability across distributed systems. MPI provides a set of functions for communication and synchronization between processes.

- **`MPI_Init()` e `MPI_Finalize()`**

  The functions `MPI_Init()` and `MPI_Finalize()` are used to initialize and terminate the MPI environment. `MPI_Init()` is called at the beginning of the MPI program to initialize the MPI environment and establish communication between processes. `MPI_Finalize()` is called at the end of the program to clean up and release resources used by the MPI environment.

- **`MPI_Allreduce()`**

  The function `MPI_Allreduce()` performs a global reduce operation on all processes in the MPI communicator and returns the result to all processes. It is useful when you need to combine the results of each process into a single variable.

- **`MPI_Reduce()`**

  The function `MPI_Reduce()` performs a reduce operation on all processes in the MPI communicator and returns the result to the root process. This is similar to `MPI_Allreduce()`, but the result is returned only to the root process.

- **`MPI_Barrier()`**

  The function `MPI_Barrier()` it is used to synchronize all processes in the MPI communicator. Each process waits until all other processes in the communicator have reached the synchronization point before continuing.

3.2 OMP (Open Multi-Processing)

OpenMP is a parallel programming API for programming on systems with shared memory, such as multicore systems. It allows you to perform operations in parallel using pragma directives within the C/C++ source code. OpenMP simplifies the parallelization of loops, code regions, and parallel tasks, allowing you to maximize the processing potential of modern multicore processors.

3.2.1 Direttive pragma

OpenMP pragma directives are special commands that are interpreted by the compiler to guide the parallelization process. The pragma directives starts with `#pragma omp` followed by a specific instruction.

- **`#pragma omp parallel`**
  The directive `#pragma omp parallel` is used to create a team of threads, where the immediately following block of code will be executed by all threads in the team.
- **`#pragma omp parallel for`**
  The directive `#pragma omp parallel for` is used to parallelize a loop, allowing threads to iterate through the loop in parallel.
  - **`#pragma omp parallel for collapse (2)`**
    The directive `#pragma omp parallel for collapse (2)` is used to parallelize two nested loops, allowing threads to iterate in parallel. This directive can improve performance when you have nested loops.
- **`#pragma omp atomic`**
  The directive `#pragma omp atomic` is used to perform an operation atomically, avoiding concurrency between threads.
- **`#pragma omp single`**
  The directive `#pragma omp single` is used to execute a block of code from a single thread, ensuring that it runs only once within a team of threads.

4 Algorithm

The clustering algorithm we chose is k-means clustering, which is a partitioning algorithm that tries to divide the data into a fixed number of groups, so that the variance within the groups is minimal.
We chose to use k-means clustering because it is a relatively simple and widely used algorithm in the clustering community. Additionally, k-means clustering is known to be highly parallelizable, making it well suited to using MPI and OMP libraries.

4.1 K-Means

The k-means algorithm is a widely used clustering method for grouping unlabeled data into a fixed number of clusters. Its goal is to divide a data set into homogeneous groups so that the points within each group are similar to each other and different from the points in the other groups.

The k-means algorithm proceeds as follows:

1. **Initialization**: Randomly initialize cluster centroids.
2. **Assigning Points to Clusters**: For each data point, calculate the distance to each centroid and assign the point to the cluster with the closest centroid.
3. **Updating centroids**: Calculate the new centroids for each cluster as the average of the points assigned to that cluster.

4. **Convergence**: Repeat steps 2 and 3 until the centroids no longer change or until a maximum number of iterations has been reached.

4.2 Parallelization of the k-means algorithm with MPI

The k-means algorithm was parallelized using the Message Passing Interface (MPI) framework to allow work to be distributed across multiple processes. Parallelization is based on the concept of partitioning data and operations between processes.
At the beginning of the algorithm, the cluster centroids are initialized randomly. In a parallel MPI environment, the zero process is responsible for initializing the centroids randomly and then distributing them to the other processes using MPI communication functions. After each process has calculated the update of the centroids using the distance to the points in the dataset, these statistics are communicated to the zero process which takes an average and performs the final update of the centroids. Once the new centroids are calculated, they are disseminated to all processes. The algorithm iterates through the steps of assigning points to clusters and updating centroids until a convergence criterion is satisfied, such as the stability of the centroids or the maximum number of iterations reached.

4.2.1 Performance evaluation

As the algorithm runs, I/O performance, k-means execution time, and total algorithm time are measured. These measurements are used to evaluate the effectiveness of parallelization with MPI in speeding up the execution of the algorithm compared to the serial version.

Parallelization with MPI allows us to effectively leverage computing resources distributed across multiple nodes, enabling greater scalability and improved performance for the k-means algorithm on large datasets. the tests were carried out using different partitions of the dataset (10% to 100% with steps of 10%) and the calculated times are then compared to evaluate how the algorithm scales with increasing data and to calculate the effective parallelization speedup and efficiency.

The speedup is calculated as follows:

$$S = \frac{T_{serial}}{T_{parallel}}$$

While the efficiency of the program in parallel:

$$E = \frac{S}{p} = \frac{T_{serial}}{T_{parallel} \cdot p}$$

where p is the number of cores available. Timings were calculated using the built-in mpi and omp functions instead of execution cluster time. the results obtained are shown in table 2. we can see how...

in addition to calculating the timing, it is possible to view the performance of the algorithm (as shown in the figure) given that the k-means algorithm does not allow us to compute precision analytically and the number of centroids in the dataset are known as they are generated by us.

4.3 Parallelization of the k-means algorithm with OMP

The k-means algorithm was parallelized using the OpenMP (OMP) framework to allow concurrent execution of operations across multiple threads. Parallelization with OMP relies on the use of special compilation directives to identify sections of code to run in parallel.
the algorithm is similar to the implementation with MPI but using instructions for OMP and therefore taking into account that the memory is shared between processes

4.4 Dataset

The dataset used as the basis for our research is the Iris dataset, one of the best known and most used datasets in the field of machine learning and data mining. The Iris dataset contains measurements on flowers of three iris species: Iris setosa, Iris versicolor and Iris virginica. For each flower, length and width measurements of the sepals and petals are provided.

However, despite the popularity of the Iris dataset, its limited size (130) can limit the effectiveness of analyzes on it, especially in contexts where greater computational complexity is required or one wants to study large-scale data.

To overcome this limitation, we developed a Python script capable of generating a new dataset. This allowed us to generate a dataset which have more than a million data points. The goal of this was to provide a larger and more diverse dataset to evaluate the effectiveness of our approach.

Furthermore, given the importance of the parameter K in the k-means algorithm, we extended our dataset by generating several versions in which the value of K varies. We created versions of the dataset with different K values, including values like 3, 6, 9. This variation allows us to explore the effect of data dimensionality on the results of the k-means algorithm and to evaluate its ability to adapt to different situations.

This idea of exploring variations in the behavior of the k-means algorithm by modifying the value of K derives from similar approaches described in other scientific papers, which have highlighted the importance of examining the behavior of the algorithm as the parameters vary to better understand the its performance and limitations.

5 Analysis

in the work carried out we carried out numerous tests and the results shown in this paper were obtained with the following parameters: k = 3, nodes = 2, GPUs = 2, . we noticed how the graphs obtained still reflect the standards seen in class. As can be seen in the graphs (1 5), efficiency tends to decrease with the increase in processes while speed up increases.
regarding the clustering results, we observed that the clusters do not vary as the number of processes/threads increases. the clustering results will be shown in the presentation .

6 Conclusion

At the end of our work, we are satisfied with the results achieved, even if our algorithm is limited in some aspects: first of all, our dataset was generated in such a way as to have the data well separated and therefore it is not very generalizable. second, we observed how the algorithm with some portions of the dataset encountered problems in calculating only some clusters. The last limitation and possible future development is the parallelization of reading and writing data on files with which we encountered several problems and which we finally decided to leave out. [0.5][h]

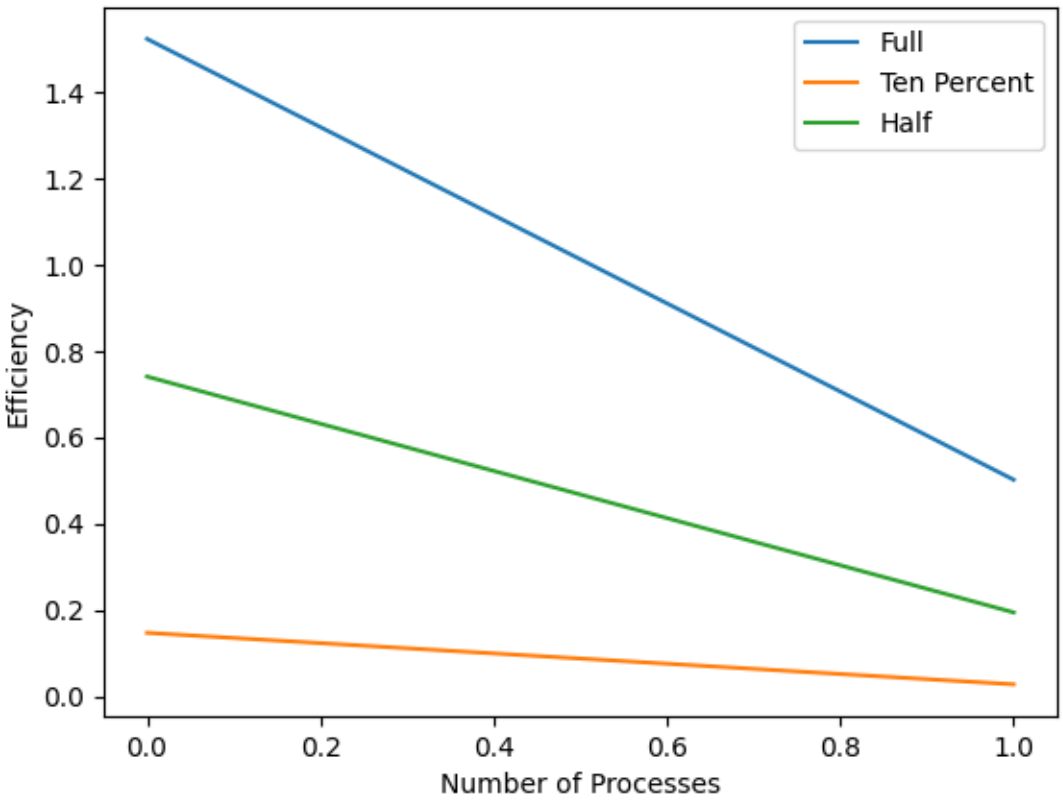

Figure 1: Efficiency for OpenMP showing the efficiency with different numbers of processes)

	10%	50%	100%
Efficiency 2 thr	0.14	0.74	1.52
Speed Up 2 thr	0.29	1.48	3.04
Speed Up 10 thr	0.27	1.94	5.01
Speed Up 10 thr	0.02	0.19	0.50

Table 1: OMP with K=3, Nodes=2, GPUs=2

	100%
Efficiency 2 thr 2 prc	0.46
Speed Up 2 thr 2 prc	1.85
Efficiency 10 thr 2 prc	0.13
Speed Up 10 thr 2	1.58

Table 2: Hybrid with K=3, Nodes=2, GPUs=2

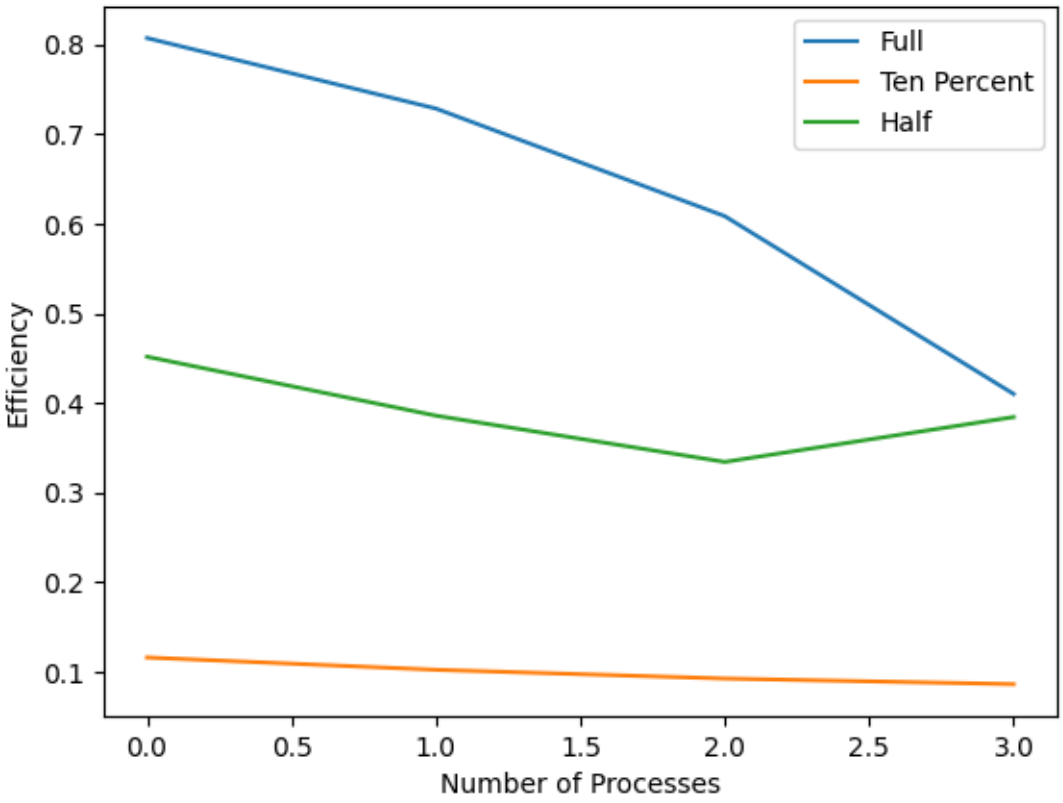

Figure 2: Efficiency for MPI showing the efficiency with different numbers of processes

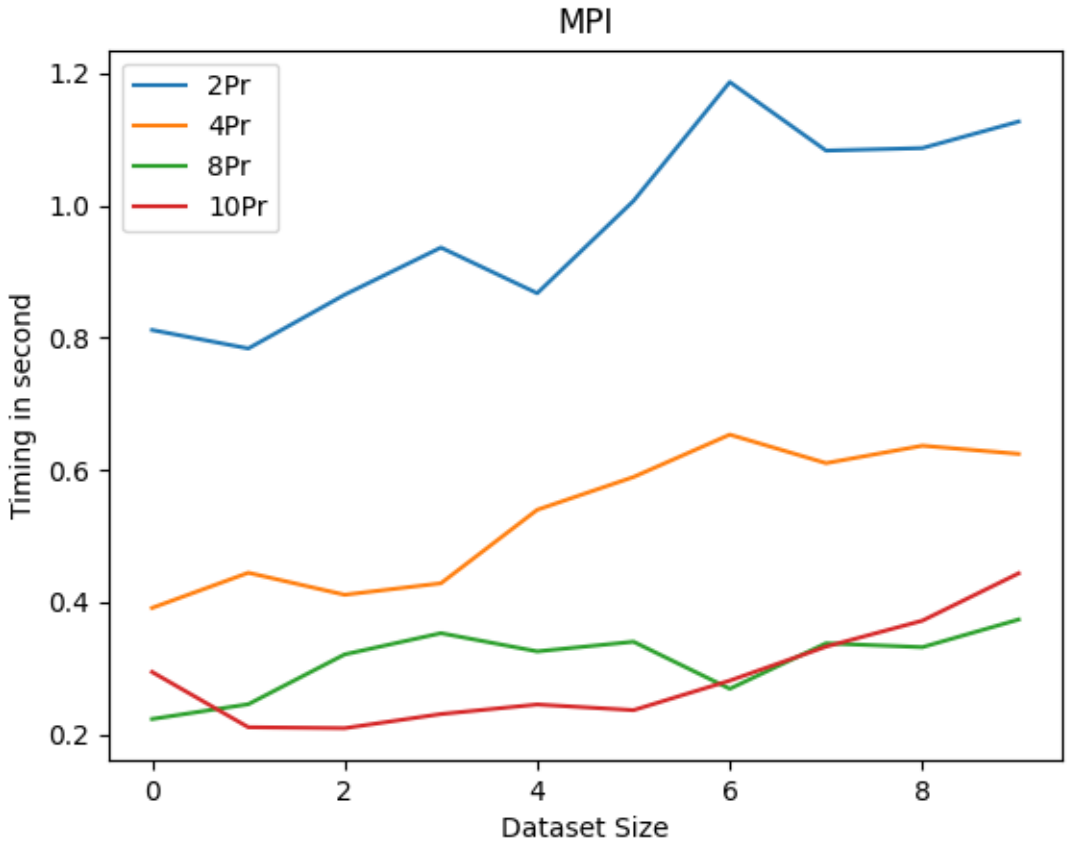

Figure 3: Timing for MPI showing how much time it takes in seconds for different datasets size (0 is 10% and 9 is 100%). pr stands for processes

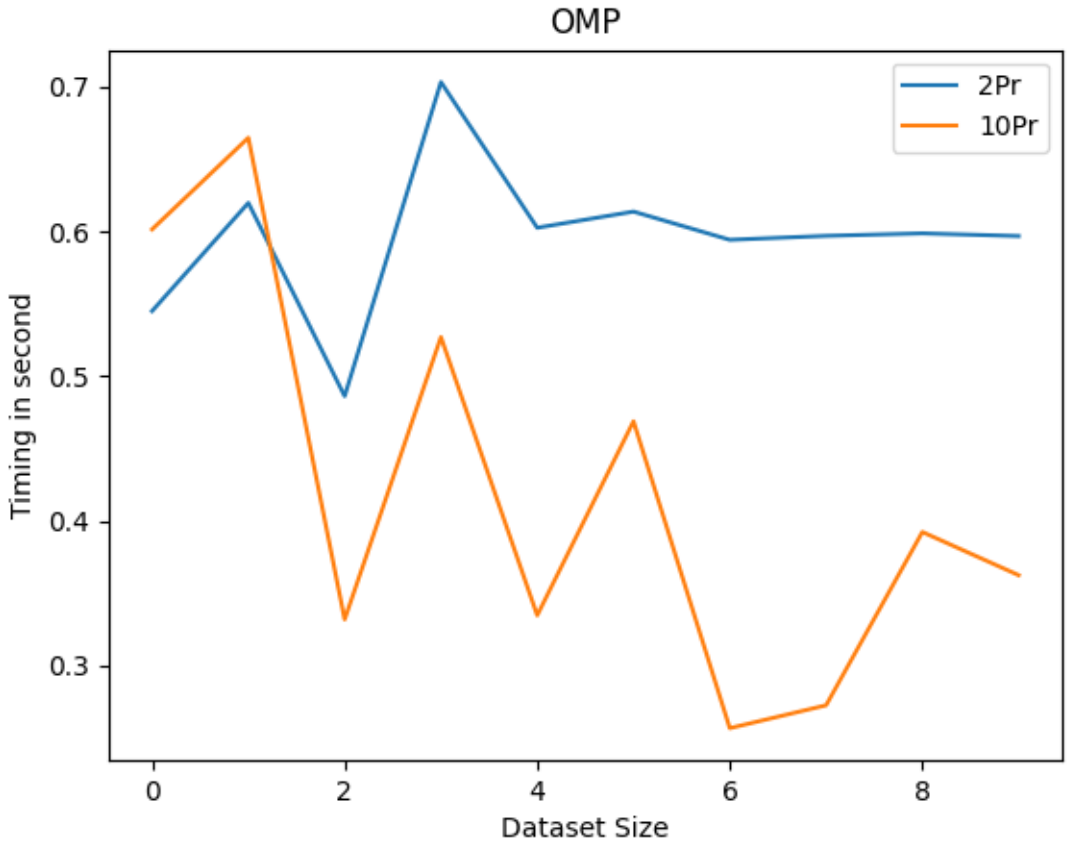

Figure 4: Timing for OMP showing how much time it takes in seconds for different datasets size (0 is 10% and 9 is 100%). , pr stands for processes

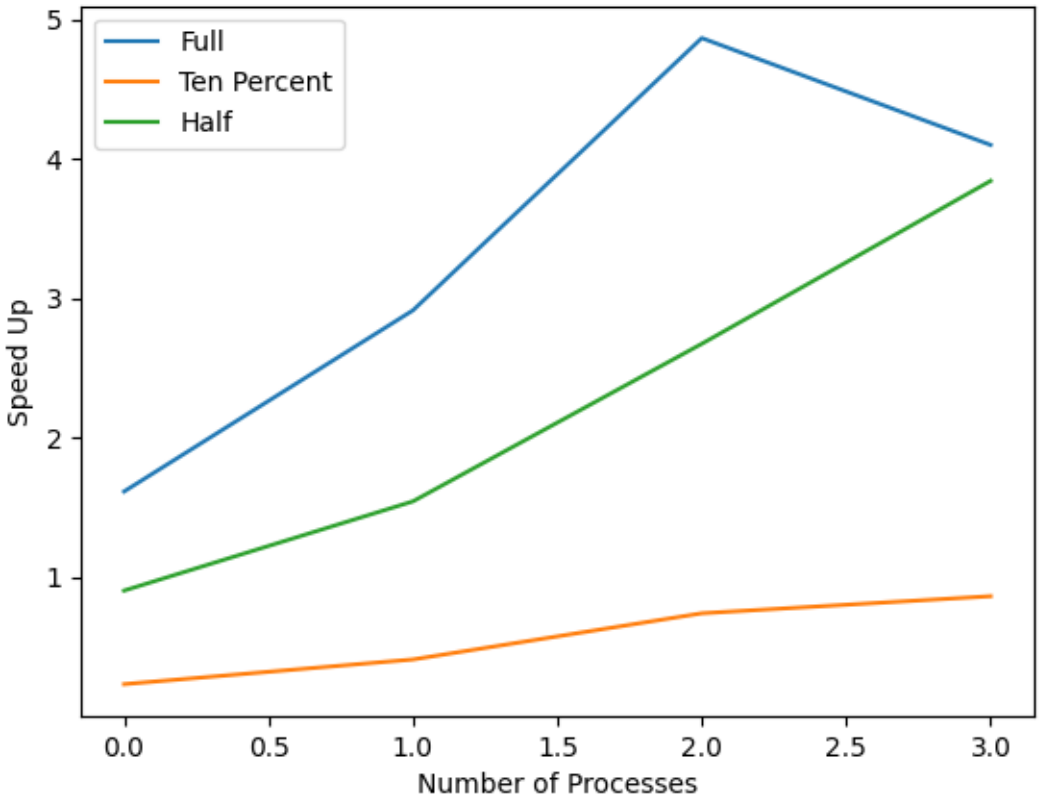

Figure 5: Speed up results varying dataset size for MPI runs

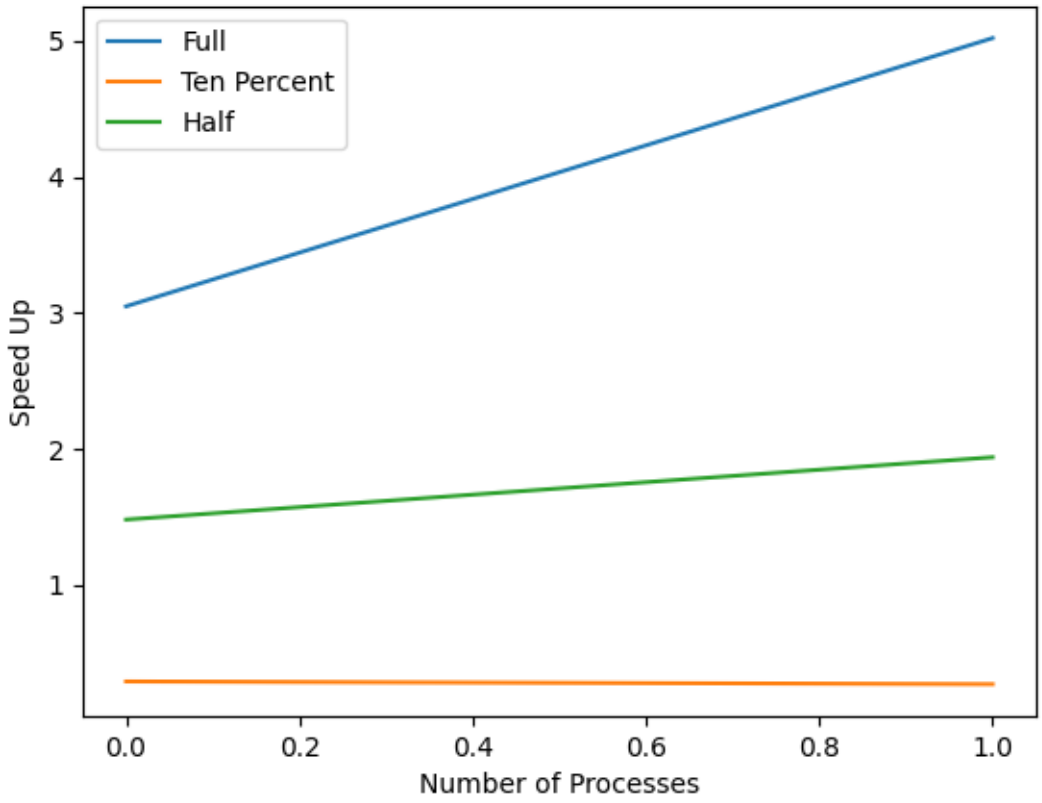

Figure 6: Speed up results varying dataset size for OMP runs

	10%	50%	100%
Efficiency 2 prc	0.12	0.45	0.81
Speed Up 2 prc	0.23	0.90	1.61
Efficiency 4 prc	0.10	0.39	0.73
Speed Up 4 prc	0.41	1.54	2.91
Efficiency 8 prc	0.09	0.33	0.61
Speed Up 8 prc	0.74	2.67	4.87
Efficiency 10 prc	0.09	0.38	0.41
Speed Up 10 prc	0.86	3.84	4.10

Table 3: MPI with K=3, Nodes=2, GPUs=2

References

[1] Zahid Ansari Ansari Abdullah Quazi Mateenuddin H. "An OpenMP Based Approach for Parallelization and Performance Evaluation of k-Means Algorithm". In: 2021.

[2] D S. Bhupal Naik, S. Deva Kumar, and S. V. Ramakrishna. "Parallel processing of enhanced K-means using OpenMP". In: *2013 IEEE International Conference on Computational Intelligence and Computing Research*. 2013, pp. 1–4. DOI: 10.1109/ICCIC.2013.6724291.

[3] Sakshi Patel, Shivani Sihmar, and Aman Jatain. "A study of hierarchical clustering algorithms". In: *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. 2015, pp. 537–541.

[4] G. Wu et al. "A Parallel K-Means Clustering Algorithm with MPI". In: *Parallel Architectures, Algorithms and Programming, International Symposium on*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2011, pp. 60–64. DOI: 10.1109/PAAP.2011.17. URL: https://doi.ieeecomputersociety.org/10.1109/PAAP.2011.17.

Algorithm 1 Parallel K-Means Algorithm with MPI

procedure KMEANSMPI($dataset$, K, max_iter)
  $centroids \leftarrow$ INITIALIZECENTROIDS($dataset$, K)
  $t1 \leftarrow$ MPI_WTIME
  $t\_per\_iter \leftarrow 0$
  $iter \leftarrow 0$
  while $iter < max\_iter$ **do**
    for i in 0 to $ROWS - 1$ **do**
      $min\_distance \leftarrow$ DBL_MAX
      $min\_index \leftarrow 0$
      for j in 0 to $K - 1$ **do**
        $distance \leftarrow$ EUCLIDEANDISTANCE(i, j)
        if $distance < min\_distance$ **then**
          $min\_distance \leftarrow distance$
          $min\_index \leftarrow j$
        end if
      end for
      if $cluster[i] \neq min\_index$ **then**
        $cluster[i] \leftarrow min\_index$
      end if
    end for
    $local\_sum[K][COLS] \leftarrow 0$
    $local\_count[K] \leftarrow 0$
    for i in $start$ to $end - 1$ **do**
      for j in 0 to $COLS - 1$ **do**
        $local\_sum[cluster[i]][j] \mathrel{+}= dataset[i][j]$
      end for
      $local\_count[cluster[i]] \mathrel{+}= 1$
    end for
    MPI_BARRIER(MPI_COMM_WORLD)
    MPI_ALLREDUCE($local\_sum$, $global\_sum$, $K * COLS$, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD)
    MPI_ALLREDUCE($local\_count$, $global\_count$, K, MPI_INT, MPI_SUM, MPI_COMM_WORLD)
    for i in 0 to $K - 1$ **do**
      for j in 0 to $COLS - 1$ **do**
        if $global\_count[i] \neq 0$ **then**
          $centroids[i][j] \leftarrow global\_sum[i][j]/global\_count[i]$
        end if
      end for
    end for
    $iter \leftarrow iter + 1$
  end while
  WRITECENTROIDSTOFILE($centroids$, K)
  $t2 \leftarrow$ MPI_WTIME
  $t\_per\_iter \leftarrow (t2 - t1)/max\_iter$
  MPI_ALLREDUCE(t_per_iter, $sum\_iter\_kmeans\_time$, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD)
end procedure

Algorithm 2 Sequential K-Means Algorithm

```
1:  procedure KMEANS(dataset, K, max_iter)
2:      centroids ← INITIALIZECENTROIDS(dataset, K)
3:      iter ← 0
4:      while iter < max_iter do
5:          for i in 0 to ROWS − 1 do
6:              min_distance ← DBL_MAX
7:              min_index ← 0
8:              for j in 0 to K − 1 do
9:                  distance ← EUCLIDEANDISTANCE(i, j)
10:                 if distance < min_distance then
11:                     min_distance ← distance
12:                     min_index ← j
13:                 end if
14:             end for
15:             if cluster[i] ≠ min_index then
16:                 cluster[i] ← min_index
17:             end if
18:         end for
19:         sum[K][COLS] ← 0
20:         count[K] ← 0
21:         for i in 0 to ROWS − 1 do
22:             for j in 0 to COLS − 1 do
23:                 sum[cluster[i]][j] += dataset[i][j]
24:             end for
25:             count[cluster[i]] += 1
26:         end for
27:         for i in 0 to K − 1 do
28:             for j in 0 to COLS − 1 do
29:                 if count[i] ≠ 0 then
30:                     centroids[i][j] ← sum[i][j]/count[i]
31:                 end if
32:             end for
33:         end for
34:         iter ← iter + 1
35:     end while
36:     WRITECENTROIDSTOFILE(centroids, K)
37: end procedure
```

Algorithm 3 OpenMP K-Means Algorithm

procedure KMEANSOMP($dataset$, K, max_iter, $sum\_iter\_kmeans\_time$)
    initialize $centroids$ **randomly**
    $iter \leftarrow 0$
    $time1, time2 \leftarrow 0$
    $count[K] \leftarrow \{0\}$
    $sum[K][COLS] \leftarrow \{\{0\}\}$
    parallel region
    while $iter < max\_iter$ **do**
        $time1 \leftarrow$ OMP_GET_WTIME
        reset $count$, sum
        parallel for
        for i in 0 to $ROWS - 1$ **do**
            $min\_distance \leftarrow$ DBL_MAX
            $min\_index \leftarrow 0$
            for j in 0 to $K - 1$ **do**
                $distance \leftarrow$ EUCLIDEANDISTANCE(i, j)
                if $distance < min\_distance$ **then**
                    $min\_distance \leftarrow distance$
                    $min\_index \leftarrow j$
                end if
            end for
            if $cluster[i] \neq min\_index$ **then**
                $cluster[i] \leftarrow min\_index$
            end if
            $count[cluster[i]] \leftarrow count[cluster[i]] + 1$
            for j in 0 to $COLS - 1$ **do**
                $sum[cluster[i]][j] \leftarrow sum[cluster[i]][j] + dataset[i][j]$
            end for
        end for
        combine $count$ and sum **across threads**
        calculate new $centroids$
        single
        for i in 0 to $K - 1$ **do**
            if $count[i] \neq 0$ **then**
                for j in 0 to $COLS - 1$ **do**
                    $centroids[i][j] \leftarrow sum[i][j]/count[i]$
                end for
            end if
        end for
        $iter \leftarrow iter + 1$
        $time2 \leftarrow$ (omp_get_wtime() $- time1)/iter$
        $*sum\_iter\_kmeans\_time \leftarrow *sum\_iter\_kmeans\_time + time2$
    end while
    write $centroids$ **to file**
end procedure